

Politechnika Świętokrzyska
Wydział Elektrotechniki, Automatyki i Informatyki

Zespół:
Kot Jarosław
Prusicki Jakub

Łamacz haseł

Projekt zespołowy
na studiach stacjonarnych
o kierunku **Informatyka**
Stopień II

Opiekun projektu:
dr inż. Paweł Paduch

Kielce, 2023

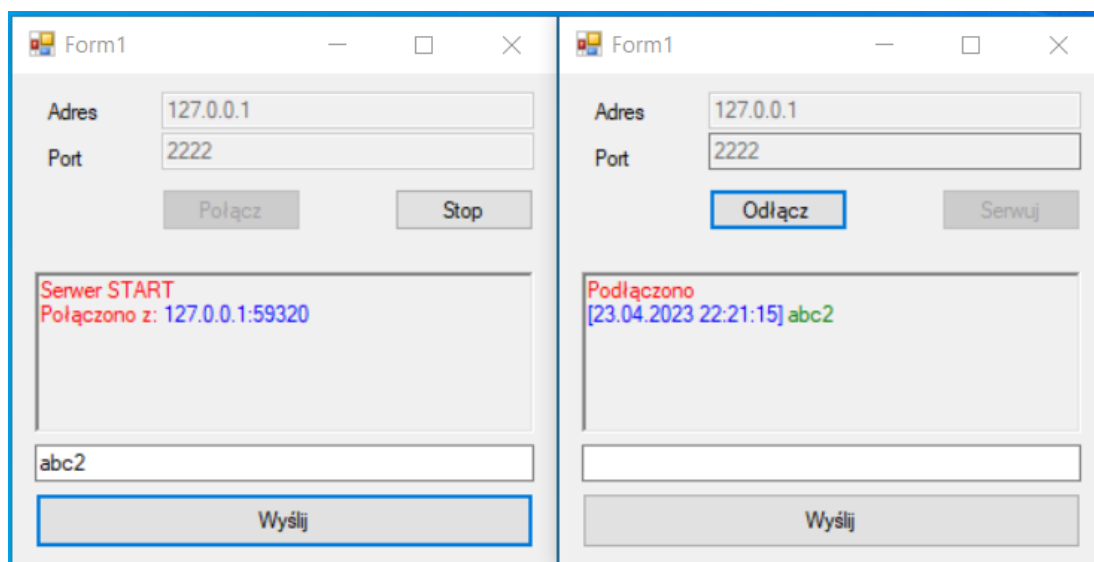
SPIS TREŚCI

1. Wstęp teoretyczny.	3
1.1. Architektura i technologie.	3
1.2. Algorytm Brute Force.	4
1.2.1. Na czym polega ?	4
1.2.2. Wady i zalety.	6
1.2.3. Wydajność.	7
1.2.4. Przykładowe rozwiązania.	7
1.3. Haszowanie.	4
1.3.1. Co to jest hash ?	4
1.3.2. MD5.	4
1.3.3. SHA-1,SHA-256.	4
1.3.4. Haszowanie “Solone”.	4
1.4. Metoda słownikowa.	4
2. Implementacja projektu.	7
2.1. Połączenie sieciowe.	3
2.2. Algorytm Brute Force.	3
2.3. Haszowanie..	3
2.4. Testowanie..	3

1. WSTĘP TEORETYCZNY.

1.1. ARCHITEKTURA I TECHNOLOGIE.

Do wykonania projektu zdecydowano się wykorzystać kod źródłowy udostępnionego już rozwiązania na platformie *achilles.tu.kielce.pl* przez opiekuna projektu. Jest to rozproszony komunikator tekstowy wykorzystujący połączeniowy protokół TCP. Zaletą tego konkretnego podejścia jest uniwersalność zaprojektowanego rozwiązania - odpowiednie przyciski *serwuj* oraz *połącz* pozwalają na opóźnienie decyzji, który z programów ma być serwerem. GUI aplikacji zostało wykonane z użyciem formatek okienkowych WinForms, będących z resztą częścią .Net Framework.



Rys. 1.1 Okienka aplikacji.

Podstawowymi elementami projektu są: struktura *Komunikat* oraz klasa *Klient*. Pozwalają one na prawidłowe przesyłanie treści w prezentowanym powyżej komunikatorze. Podczas wykonywania prac nad rozpraszaniem to właśnie ich modyfikacje będą miały największe znaczenie.

```
[Serializable]
public struct Komunikat
{
    public string tresc;
    public bool wazna;
    public string nadawca;
    public DateTime czasNadania;
    public DateTime czasOdbioru;
}
```

Rys. 1.2. Struktura komunikat.

```
class Klient
{
    public Thread watek;
    public TcpClient tcpKlient;
}
```

Rys. 1.3. Klasa Klient.

1.2. ALGORYTM BRUTE FORCE.

1.2.1 NA CZYM POLEGA ?

Algorytm siłowy (z ang. **brute force**) jest to taki algorytm, którego działanie można streścić następująco: polega on na sukcesywnym sprawdzaniu wszystkich możliwych kombinacji, w poszukiwaniu rozwiązania problemu. Dla niektórych klas problemów, jest to podejście, pod pewnymi warunkami, jedyne niezawodne. Stosując takie podejście w programowaniu, rozwiązujemy problem przez weryfikację i ocenę wszystkich wariantów postępowania. Gdybyśmy dysponowali, dowolnie długim czasem na poszukiwanie rozwiązania oraz zasobami sprzętowymi (na przykład nieskończenie dużą pamięcią operacyjną lub przestrzenią dyskową), to algorytm taki zakończy się zawsze powodzeniem. O związkach metody brute force, z kombinatoryką wspomniano nie bez przyczyny. Korzystając z kombinatoryki, można policzyć ile jest możliwych takich haseł. Po obliczeniach można stwierdzić, że wartość tego zagadnienia jest sumą wariacji z powtórzeniami. Dla przypomnienia, wariację z powtórzeniami wykorzystujemy wtedy, gdy chcemy wiedzieć ile możemy stworzyć różnych układów k -elementowych, mając do dyspozycji n -elementów, przy czym kolejność elementów w układzie jest istotna, a elementy mogą się powtarzać^[1].

Niech:

n – ilość liter w słowniku,

k – ilość znaków w haśle,

N – ilość możliwych haseł

$$\begin{aligned} N &= \sum_{k=1}^{k=n} W_n^k = W_n^1 + W_n^2 + \dots + W_n^{n-1} + W_n^n = \\ &= n^1 + n^2 + \dots + n^{n-1} + n^n \end{aligned}$$

Rys. 1.1.1 Wzór na liczbę kombinacji

Innymi słowy metoda siłowa rozwiązywania jakiegoś zadania polega na wyczerpaniu wszystkich możliwości. Dla anagramów oznacza to wygenerowanie listy wszystkich możliwych łańcuchów ze znaków łańcucha s_1 i sprawdzenie, czy s_2 znajduje się na tej liście. Nie jest to jednak zalecane podejście, przynajmniej w tym przypadku.

Zauważmy mianowicie, że dla ciągu znaków s_1 o długości n mamy n wyborów pierwszego znaku, $(n-1)$ możliwości dla znaku na drugiej pozycji, $(n-2)$ na trzeciej pozycji itd. Musimy zatem wygenerować $n!$ łańcuchów znaków. Dla ustalenia uwagi przyjmijmy, że s_1 składa się z 20 znaków. Oznacza to konieczność wygenerowania

In [3]:

```
import math
math.factorial(20)
```

Out[3]:

2432902008176640000

Rys. 1.1.2 Ilość możliwych łańcuchów.

łańcuchów znaków, a następnie odszukanie wśród nich ciągu s_2 . Można było dostrzec takie podejście już wcześniej - szczególnie, jeśli chodzi o to ile czasu zajmuje algorytm klasy $O(n!)$, dlatego nie jest to polecane podejście do zagadnienia anagramów^[2].

1.2.2 WADY I ZALETY

Zalety:

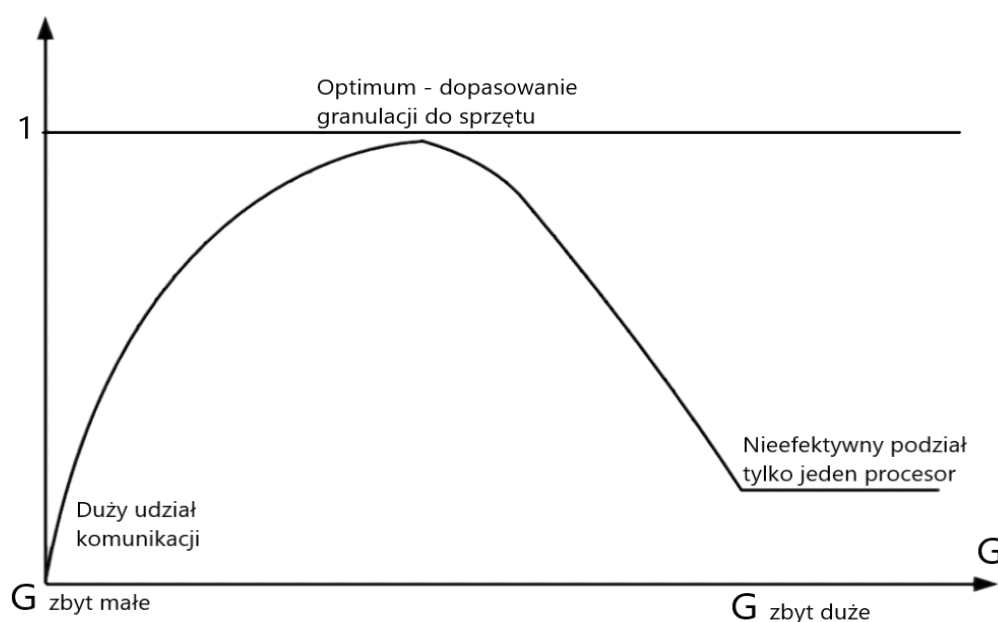
- ❖ łatwość zrozumienia i zaimplementowania
- ❖ uniwersalność - algorytm może być stosowany do rozwiązywania różnego rodzaju problemów, w tym problemów wielowymiarowych
- ❖ nie wymaga dodatkowych założeń
- ❖ możliwość stosowania nawet w przypadku problemów nieliniowych

Wady:

- ❖ mniejsza dokładność w porównaniu do innych metod numerycznych
- ❖ wymaga dużego nakładu obliczeniowego
- ❖ wymaga wyznaczenia dokładnej funkcji sił, która może być trudna do znalezienia w niektórych przypadkach

1.2.3 WYDAJNOŚĆ

Ziarnistość projektu to poziom podziału pracy w projekcie, który określa, na jakie mniejsze części jest podzielony projekt i jakie zależności występują między nimi. W kontekście programowania współbieżnego, ziarnistość projektu odnosi się do sposobu podziału pracy pomiędzy wątki lub procesy. Wydajność ziarnistości projektu zależy przede wszystkim od rodzaju zadania oraz architektury systemu, w którym jest realizowane. W ogólnym przypadku, mniejsza ziarnistość (czyli mniejsze podziały pracy) może przyspieszyć wykonanie zadania, ale może też wprowadzić koszty synchronizacji i komunikacji między wątkami lub procesami. Z kolei większa ziarnistość (większe podziały pracy) może zmniejszyć koszty synchronizacji i komunikacji, ale może też wprowadzić koszty narzutu, związanego z tworzeniem i zarządzaniem dodatkowymi wątkami lub procesami. W przypadku projektów o dużym obciążeniu procesora, mniejsza ziarnistość jest zwykle bardziej wydajna, ponieważ zapewnia lepsze wykorzystanie zasobów sprzętowych. W przypadku projektów o dużym obciążeniu sieciowym lub wejścia-wyjścia, większa ziarnistość może być bardziej wydajna, ponieważ zmniejsza ilość operacji synchronizacji i komunikacji. Ostatecznie, wybór odpowiedniej ziarnistości projektu zależy od indywidualnych wymagań projektu oraz dostępnych zasobów sprzętowych. Warto pamiętać, że optymalna ziarnistość projektu może być osiągnięta tylko poprzez testowanie i analizę wyników działania systemu w różnych konfiguracjach.



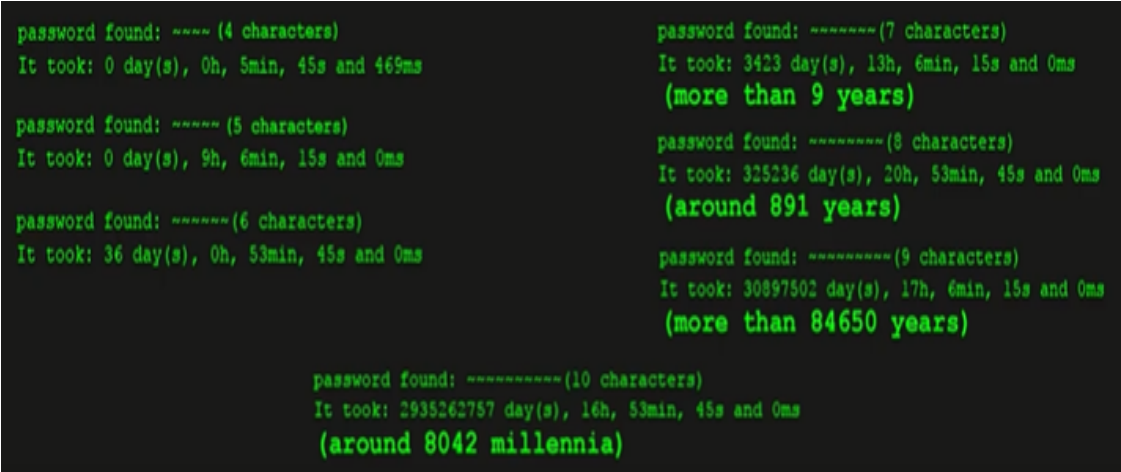
Rys. 2.4.1 Wykres doboru ziarnistości^[4].

W ogólnym przypadku ziarnistość kodu jest pojęciem względnym, ściśle związanym z określoną architekturą systemu wieloprocessorowego. Kod wykonany na systemie z pamięcią dzieloną sklasyfikowany jako gruboziarnisty, może zostać uznany za drobnoziarnisty w systemie z pamięci rozproszoną. Zgodnie z tym, co zostało przedstawione, w zależności od architektury koszt zarządzania wielozadaniowością (np. zarządzanie wątkami) może znacznie różnić się od siebie. W zależności od typu użytej transformacji pętli możliwe jest uzyskanie podanej ziarnistości kodu dla wybranej architektury. Tabela 2.4.2 przedstawia podział transformacji pętli programowych w zależności od możliwej do uzyskania ziarnistości.

Transformacja	Ziarnistość ¹	
	drobno-	grubo-
Zmiana kolejności wykonania pętli (ang. <i>loop interchange</i>)	●	●
Rozszerzenie skalaru (ang. <i>scalar expansion</i>)	●	○
Zmiana nazw zmiennych skalarnych (ang. <i>scalar renaming</i>)	●	○
Zmiana nazw zmiennych tablicowych (ang. <i>array renaming</i>)	●	○
Podział węzłów (ang. <i>node splitting</i>)	●	○
Redukcja (ang. <i>reduction</i>)	●	○
Podział zmiennych indeksowych (ang. <i>index-set splitting</i>)	●	○
Przekoszenie pętli (ang. <i>loop skewing</i>)	●	●
Prywatyzacja (ang. <i>privatization</i>)	○	●
Podział pętli (ang. <i>loop distribution</i>)	○	●
Wyrównanie (ang. <i>alignment</i>)	○	●
Replikacja kodu (ang. <i>code replication</i>)	○	●
Łączenie pętli (ang. <i>loop fusion</i>)	○	●
Odwrócenie wykonania pętli (ang. <i>loop reversal</i>)	○	●
Wielowymiarowe łączenie pętli (ang. <i>multilevel loop fusion</i>)	○	●

Rys. 2.4.1 Tabela podziału ziarnistości^[7822].

Tempo łamania hasła zależy od wielu czynników, takich jak długość hasła, użyte znaki, algorytm szyfrowania, moc obliczeniowa używanego sprzętu, itp. Oczywiście, im dłuższe i bardziej złożone hasło, tym dłużej będzie trwać proces łamania za pomocą brute force. Dla przykładu, hasło składające się z 6 małych liter (26 możliwości na każdej pozycji) może zostać złamane w ciągu kilku minut na zwykłym komputerze. Natomiast, hasło składające się z 12 znaków (liczby, litery, znaki specjalne) może wymagać lat pracy dla najpotężniejszych superkomputerów.



```
password found: ~~~~ (4 characters)
It took: 0 day(s), 0h, 5min, 45s and 469ms

password found: ~~~~~ (5 characters)
It took: 0 day(s), 9h, 6min, 15s and 0ms

password found: ~~~~~~ (6 characters)
It took: 36 day(s), 0h, 53min, 45s and 0ms

password found: ~~~~~~ (7 characters)
It took: 3423 day(s), 13h, 6min, 15s and 0ms
(more than 9 years)

password found: ~~~~~~ (8 characters)
It took: 325236 day(s), 20h, 53min, 45s and 0ms
(around 891 years)

password found: ~~~~~~ (9 characters)
It took: 30897502 day(s), 17h, 6min, 15s and 0ms
(more than 84650 years)

password found: ~~~~~~ (10 characters)
It took: 2935262757 day(s), 16h, 53min, 45s and 0ms
(around 8042 millennia)
```

Rys. 2.4.1 Przybliżony przedział czasowy dla algorytmu BruteForce ^[7822].

1.2.4 PRZYKŁADOWE ROZWIĄZANIA

Aby wybrać odpowiednią wersję algorytmu, sprawdzono dotychczasowe rozwiązania opublikowane już w zasobach internetu. Najważniejsze z nich, które pomogły w zbudowaniu algorytmu pochodzą z następujących źródeł:

- Dot Net Office^[1]
- Użytkownik serwisu github.com *jwoschitz*^[2]
- C # Corner^[3]

Oczywistym jest fakt, że nasz kod będzie nadal modyfikowany, będą implementowane dodatkowe funkcje, a te istniejące rozwijane, jednak na ten moment efekty prostego i poprawnie działającego algorytmu wystarczyły aby przetestować program przesyłający pojedyncze hasło od klienta do serwera aby ten mógł je złamać i zapisać do opisywanego już wcześniej pliku.

1.3. HASZOWANIE.

1.3.1 Co to jest HASH ?

Hash'em nazywamy wynik działania operacji matematycznej (nazywanej funkcją skrótu) na określonym ciągu znaków (np. na hasle lub pliku). Funkcja ta przekształca podane przez użytkownika dane wejściowe (np. hasło) na krótką, posiadającą stały rozmiar wartość znakową. Ważną własnością funkcji skrótu jest to, że jest ona nieodwracalna. Osoba mając hash nie może zastosować funkcji odwrotnej do funkcji skrótu by poznać ciągu znaków (np. hasła), dla którego dany hash został wygenerowany. Haszowanie jest to pewna technika rozwiązywania ogólnego problemu słownika. Przez problem słownika rozumiemy tutaj takie zorganizowanie struktur danych i algorytmów, aby można było w miarę efektywnie przechowywać i wyszukiwać elementy należące do pewnego dużego zbioru danych (uniwersum). Przykładem takiego uniwersum mogą być liczby lub napisy (wyrazy) zbudowane z liter jakiegoś alfabetu^[4]. Ogólnie rzecz biorąc hash to wynik zastosowania funkcji skrótu na pewnym ciągu znaków (hasle lub pliku). Hash pozostaje stały dla danego pliku, a każda choćby najmniejsza modyfikacja jednego znaku powoduje niezgodność nowo zmodyfikowanego pliku ze starym hashem.^[1]

Przykład:

Po zastosowaniu funkcji skrótu MD5 na hasle **AlaMaKota1234!** otrzymamy wartość wyjściową w postaci ciągu znaków **e3f52f31c9bd800b9724ced17ba8be96**.

1.3.2 MD5.

Mimo popularności MD5 nie jest już uważane za funkcję w pełni bezpieczną – opracowano wiele technik znajdujących kolizję tej funkcji. Jedną z nich jest metoda *MD5 Tunneling* znajdująca kolizję w czasie poniżej 10 sekund, wykorzystując do tego moc laptopa przeciętnej klasy.

Odkrycia tego rodzaju oczywiście są ciekawe, ale w praktyce nie wpływają znacznie na bezpieczeństwo hashy MD5. Większość technik znajdowania kolizji podaje na wejściu MD5 dane binarne, a więc dane ze zdecydowanie szerszego zakresu niż znaki ASCII.

Chociaż dzisiaj nie istnieją efektywne metody znajdujące kolizje w hashach MD5 haseł, odradza się stosowanie tej funkcji. Istnieją realne przesłanki wskazujące, że sytuacja ta zmieni się w najbliższych latach.

```

public static string CreateMD5(string input)
{
    using (System.Security.Cryptography.MD5 md5 = System.Security.Cryptography.MD5.Create())
    {
        byte[] inputBytes = System.Text.Encoding.ASCII.GetBytes(input);
        byte[] hashBytes = md5.ComputeHash(inputBytes);

        //return Convert.ToHexString(hashBytes); // .NET 5 +

        // Convert the byte array to hexadecimal string
        StringBuilder sb = new System.Text.StringBuilder();
        for (int i = 0; i < hashBytes.Length; i++)
        {
            sb.Append(hashBytes[i].ToString("X2"));
        }
        return sb.ToString().ToLower();
    }
}

```

Rys. 2.4.1 Przykładowy kod haszowania za pomocą MD5 ^[7822].

1.3.3 SHA-1,SHA-256.

SHA1 jest obecnie największym konkurentem MD5. Istnieje kilka ataków teoretycznych na tę funkcję, jednak w praktyce nie zagrażają one jeszcze bezpieczeństwu hashy. Jednak tego rodzaju sytuacje sugerują, że warto zainteresować się nowszą wersją algorytmu.

SHA2 jest w tej chwili jedną z najbezpieczniejszych wersji algorytmów rodziny SHA. SHA2 istnieje w czterech odmianach: SHA-224, SHA-256, SHA-384 oraz SHA-512. Wszystkie warianty działają w podobny sposób, jednak używają struktur danych o różnej wielkości oraz zwracają hash różnej długości. Niestety funkcje z tej grupy nie są często stosowane, ponieważ ich użycie często wymaga kompilacji dodatkowych modułów (np. do baz danych). Dodatkowym powodem było też oczekiwanie na finalną wersję algorytmu SHA3, który używa nowego podejścia – więc teoretycznie ataki na wcześniejsze wersje SHA nie powinny wpływać na jego bezpieczeństwo. Niestety – dalej jest to algorytm bardzo szybki.

Różnice pomiędzy konkretnymi algorytmami z rodziny SHA przedstawia poniższa tabela.

Secure Hash Algorithm Family

Algorytm i wariant		Rozmiar wyjścia (bity)	Wewnętrzny rozmiar stanu (bity)	Max rozmiar wiadomości (bity)	Znalezione Kolizje	Przykładowa wydajność (MB/s)	
						32b	64b
SHA-0		160	160	$2^{64}-1$	Tak	-	-
SHA-1		160	160	$2^{64}-1$	Teoretyczny atak (2^{51})	153	192
SHA-2	SHA-256/224	256/224	256	$2^{64}-1$	Brak	111	139
	SHA-512/384	512/384	512	$2^{128}-1$	Brak	99	154

Rys. 2.3.1 SHA.

Natomiast poniżej przedstawiono obrazowy przykład działania programu ze strony <https://sha256algorithm.com> z wykorzystaniem tego sposobu haszowania (w wersji SHA-256) dla wygenerowania unikalnego hasza do hasła **AlaMaKota1234!**

The screenshot displays a detailed visualization of the SHA-256 algorithm's internal state and operations. The interface is divided into several sections:

- Text Input:** The input string "AlaMaKota1234!" is entered in the top-left field.
- Message schedule - 1st chunk:** This section shows the iterative processing of the message. It includes:
 - Update working variables as:** Calculations for $h = g$, $g = f$, $f = e$, $e = d + \text{Temp1}$, $d = c$, $c = b$, $b = a$, and $a = \text{Temp1} + \text{Temp2}$.
 - Where:** Definitions for $\text{Temp1} = h + \text{I1} + \text{Choice} + k53 + w63$ and $\text{Temp2} = \text{I0} + \text{Majority}$.
 - Choice:** Calculations for I1 (XOR of rightrotated e by 6, 11, and 25 bits) and Choice (XOR of e and f).
 - Majority:** Calculations for I0 (XOR of rightrotated a by 2, 13, and 22 bits) and Majority (XOR of a , b , and c).
- 2. Add the working variables to the current hash value:** Calculations for $h0 = h0 + a$, $h1 = h1 + b$, $h2 = h2 + c$, $h3 = h3 + d$, $h4 = h4 + e$, $h5 = h5 + f$, $h6 = h6 + g$, and $h7 = h7 + h$.
- 3. Append hash values to get final digest:** The final SHA256 hash is calculated as $h0 \parallel h1 \parallel h2 \parallel h3 \parallel h4 \parallel h5 \parallel h6 \parallel h7$.
- Working Variables:** A list of variables a through h and k constants, each represented by a 256-bit binary string.
- K constants:** A list of 64 constant values $k0$ through $k63$, each represented by a 256-bit binary string.

The visualization uses a color-coded system to highlight specific parts of the algorithm, such as the message schedule (blue), working variables (green), and constants (red).

Rys. 2.3.2 SHA.^[3]

```

public class Program
{
    public static void Main1111()
    {
        string password = "test";
        using (SHA256 sha256Hash = SHA256.Create())
        {
            string hash = GetHash(sha256Hash, password);

            Console.WriteLine($"The SHA256 hash of '{password}' : {hash}.");

            Console.WriteLine("Verifying the hash...");

            if (VerifyHash(sha256Hash, password, hash))
            {
                Console.WriteLine("The hashes are the same.");
            }
            else
            {
                Console.WriteLine("The hashes are not same.");
            }
        }
        Console.ReadLine();
    }

    private static string GetHash(HashAlgorithm hashAlgorithm, string input)
    {
        byte[] data = hashAlgorithm.ComputeHash(Encoding.UTF8.GetBytes(input));

        var sBuilder = new StringBuilder();

        for (int i = 0; i < data.Length; i++)
        {
            sBuilder.Append(data[i].ToString("x2"));
        }

        return sBuilder.ToString();
    }

    private static bool VerifyHash(HashAlgorithm hashAlgorithm, string input, string hash)
    {
        var hashOfInput = GetHash(hashAlgorithm, input);

        StringComparer comparer = StringComparer.OrdinalIgnoreCase;

        return comparer.Compare(hashOfInput, hash) == 0;
    }
}

```

Rys. 2.3.3 Przykładowa implementacja SHA-256.

1.3.4 HASHOWANIE SOLONE.

Hash "solony" to technika używana w kryptografii i informatyce, która polega na dodaniu losowej wartości (tzw. "soli") do hasła przed jego haszowaniem. W procesie solenia, losowa wartość jest dodawana do hasła, a następnie całość jest poddawana funkcji skrótu (hash), takiej jak np. SHA-256. Wynikowy skrót jest zapisywany w bazie danych razem z wartością soli, która była użyta do solenia hasła. Dzięki temu, nawet jeśli dwa użytkownicy używają tego samego hasła, to ich solone hasła w bazie danych będą się od siebie różnić, ponieważ soli użyte do ich solenia będą różne. Dzięki temu, złamanie jednego hasła nie oznacza złamania wszystkich solonych haseł w bazie danych. Haszowanie solone jest jednym z podstawowych sposobów przechowywania haseł i jest stosowane w wielu systemach logowania i uwierzytelniania, w celu zwiększenia bezpieczeństwa przechowywanych haseł.

1.4. METODA SŁOWNIKOWA.

Ataki słownikowe niezmiennie pozostają jedną z najskuteczniejszych metod odzyskiwania haseł. Tam, gdzie próby siłowe nie są w stanie w rozsądnym czasie doprowadzić do jakichkolwiek rezultatów, dobry słownik może zdziałać cuda. Do skutecznego działania niezbędne jest odpowiednie oprogramowanie oraz właśnie dobry słownik. O ile programy takie jak John The Ripper czy hashcat są dobrze znane i łatwo dostępne, o tyle znalezienie dobrego słownika nie jest już wcale takie proste. W celu odzyskania haseł z odpowiadających im skrótów (hashy) możemy skorzystać z odpowiedniego oprogramowania. Możemy też odwiedzić jeden z wyspecjalizowanych serwisów internetowych. Często są one w stanie w ciągu kilku sekund odzyskać hasła ze skrótów wyznaczonych za pomocą algorytmów takich jak: LM, NTLM, md2, sha224, ripeMD160, whirlpool, MySQL 4.1+. Do wykonania powyższego zadania serwisy wykorzystują ogromne tablice skrótów i odpowiadających im haseł o wielkościach liczonych w setkach gigabajtów. Odpowiednie tablicowanie wpływa na szybkość całego procesu, zaś o skuteczności decyduje jakość oraz rozmiar bazowego słownika^[3]

Słownikowe algorytmy kodowania polegają na kolejnym czytaniu sekwencji danych wejściowych i przeszukiwaniu słownika w dokładnie tej samej sekwencji danych. W przypadku zakończenia przeglądu sukcesem, indeks właściwej pozycji słownika staje się reprezentacją wyjściową przy czym im dłuższa jest sekwencja tym efektywniejszą jest kompresja.

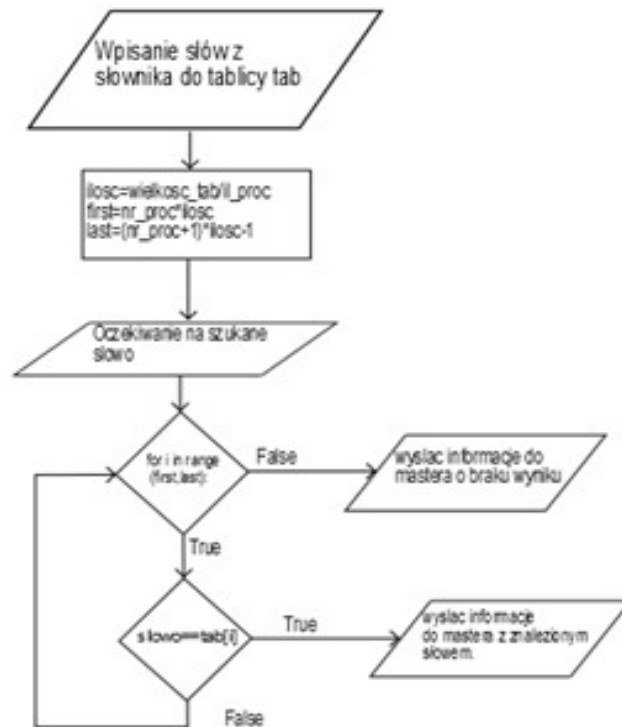
W kodowaniu słownikowym można także zastosować statyczny model słownika. Jest on skuteczniejszy, gdyż zna z wyprzedzeniem pojawiające się ciągi danych. Problemy związane z koniecznością dopisania do zbioru danych skompresowanych słownika użytego przy kodowaniu, są identyczne jak w metodach statycznych. Są stosowane w praktyce, ale jedynie do określonych przypadków kodowania w miarę stacjonarnych zbiorów danych o charakterystyce znanej a priori. Algorytmy przeznaczone do ogólnych zastosowań są w zdecydowanej większości adaptacyjne. W statycznym (entropijnym) kodowaniu pojedynczy symbol zastępowany jest ciągiem bitów o zmiennej długości. W przypadku kodowania słownikowa koncepcja jest odwrotna. Ciąg symboli o zmiennej długości jest zastępowany sekwencją kodową - indeksem w tablicy słownika. Ogólnie rzecz biorąc model ten polega na budowaniu słownika na podstawie znaków alfabetu, bądź to uzupełniany o słowa oparte na analizie danych pochodzących ze zbiorów kompresji. Następnie słownik ten jest zapisywany, bądź przesyłany do dekodera.^[4]

Algorytm^[4]:

1. Inicjacja słownika - pierwsza pozycja jako NULL.
2. Przeszukaj aktualny słownik i ustal najdłuższy łańcuch z kolejno czytanych symboli zgodny ze słownikiem.
3. Utwórz słowo kodowe najdłuższego łańcucha: indeks + kolejny symbol.
4. Dopisz nową frazę do słownika (z pkt.3).
5. Powtarzaj od pkt.2 aż do zakodowania wszystkich symboli na wejściu.

Algorytm łamania haseł metodą słownikową jest techniką używaną do siłowego odgadywania haseł do systemów. Jego realizacja wymaga dużych nakładów obliczeniowych, dlatego też uzasadnione jest wykorzystanie do tego celu programowania równoległego. W artykule przedstawiono i porównano ze sobą, równoległe implementację tego algorytmu w trzech różnych środowiskach programowania równoległego, realizujące zrównoleglenie obliczeń na poziomie danych oraz środowisko procesorów kart graficznych^[5].

Nie zawsze odzyskiwanie czy też testowanie siły haseł sprowadza się do łamania hashy. Wiele programów lub metod słownikowego odzyskiwania haseł wymaga podania wprost słownika haseł, które chcemy po kolei sprawdzić. Przykładem mogą być tu ataki online, czyli takie, które próbują odzyskać hasło na podstawie przeprowadzania kolejnych prób logowania do docelowego systemu/usługi.^[3]

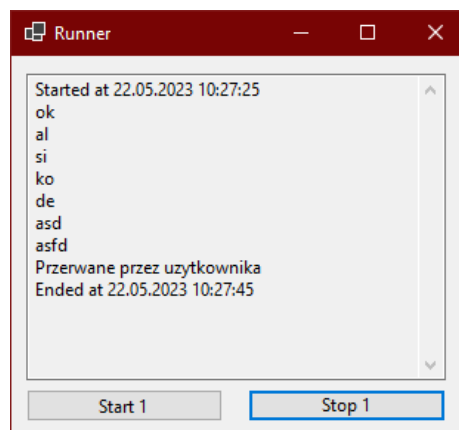


Rys. 1.2.1 Schemat blokowy algorytmu słownikowego.

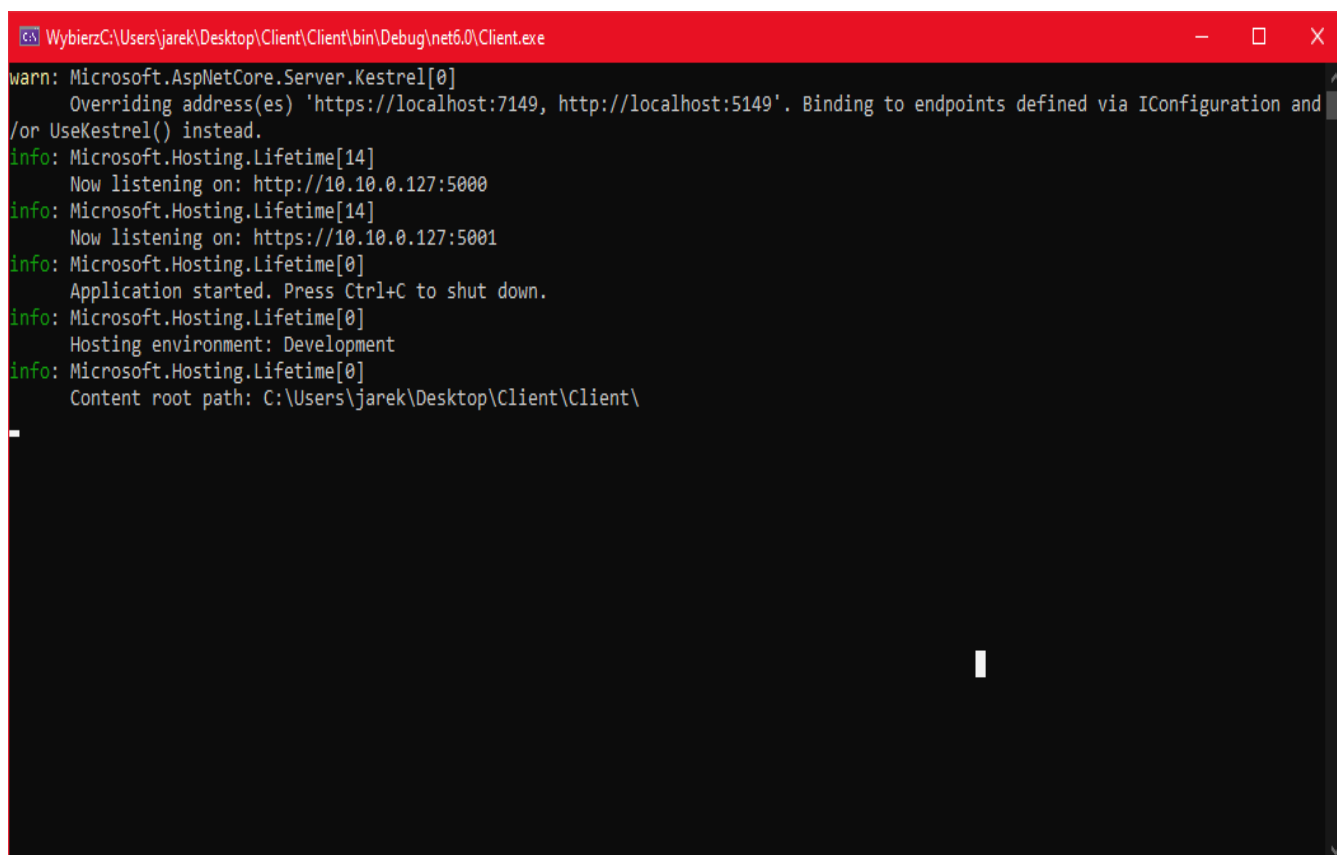
2. IMPLEMENTACJA PROJEKTU.

2.1 POŁĄCZENIE SIECIOWE.

Aplikacja “Łamacz Hasel” stworzona w języku C# ma za zadanie łamania haseł za pomocą serwera wysyłającego hasła oraz klientów, którzy łamią te hasła. Program dzieli się na dwie części “Serwer” oraz “Klient”. Pierwsza część służy do obsługi klientów oraz haseł. Metoda BtnStart1_Click jest wywoływana po kliknięciu przycisku **"Start"**. Czyści pole tekstowe textBox1, a następnie dodaje wiadomość o rozpoczęciu operacji oraz odczytuje zawartość pliku "hasla.txt".



Następnie zawartość pliku jest podzielona na bloki i przekazana do metody `CallToClients` w celu wywołania żądań HTTP na serwerze. Wykorzystywana jest klasa `Task` do tworzenia zadań asynchronicznych, a metoda `Parallel.ForEach` jest używana do uruchomienia zadań równolegle. Na końcu oczekuje się na zakończenie wszystkich zadań asynchronicznych i wyświetla wiadomość o zakończeniu operacji. Metoda `CallToClients` wykonuje żądanie HTTP na podany adres URL. Tworzony jest klient `HttpClient` z zdefiniowanym obsługiwaniem certyfikatów klienta i serwera. Następnie wywoływane jest żądanie GET na podany adres URL, a odpowiedź jest odczytywana asynchronicznie. Jeśli odpowiedź jest udana (kod statusu 200), odczytywana jest treść odpowiedzi. Następnie odpowiedź i URL są dodawane do słownika `Catalog`. Kolejnym krokiem jest przygotowanie danych do żądania POST, które są zserializowane do formatu JSON. Nagłówki żądania są ustawiane na wartość akceptującą dane w formacie JSON. Następnie wywoływane jest żądanie POST na podany adres URL, a odpowiedź jest odczytywana asynchronicznie. Jeśli odpowiedź jest udana, deserializowany jest obiekt JSON do klasy `Model`, a wartość `Message` jest dodawana do pola tekstowego `textBox1`. Metoda `BtnStop1_Click` jest wywoływana po kliknięciu przycisku "Stop". Wykonuje żądanie POST na wszystkich adresach URL zapisanych w słowniku `Catalog` w celu zakończenia działania klientów.



```
WybierzC:\Users\jarek\Desktop\Client\Client\bin\Debug\net6.0\Client.exe
warn: Microsoft.AspNetCore.Server.Kestrel[0]
      Overriding address(es) 'https://localhost:7149, http://localhost:5149'. Binding to endpoints defined via IConfiguration and/or UseKestrel() instead.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://10.10.0.127:5000
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://10.10.0.127:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\jarek\Desktop\Client\Client\
```

Obraz.6.5 przykładowy wygląd po uruchomieniu klienta.

Druga część aplikacji to klient w którym łamane są hasła za pomocą "BruteForce". Dzieli się on na "Service" oraz "Controller". Klasa BruteForceService posiada pole m_TokensCatalog, które jest słownikiem przechowującym identyfikatory żetonów (tokenów) generowane za pomocą klasy Guid oraz odpowiadające im obiekty CancellationTokenSource. Metoda GetToken() generuje nowy żeton (token) za pomocą klasy Guid, tworzy nowy obiekt CancellationTokenSource, dodaje żeton i odpowiadający mu obiekt CancellationTokenSource do słownika m_TokensCatalog i zwraca wygenerowany żeton jako wartość typu string. Metoda BruteForce(string[] passwords, string guid) przyjmuje tablicę haseł do przeprowadzenia ataku brute force oraz identyfikator żetonu (tokenu) jako parametry. Na podstawie identyfikatora żetonu, metoda pobiera odpowiadający mu obiekt CancellationToken z m_TokensCatalog i wywołuje metodę BruteForce(string[] passwords, CancellationToken cancellationToken). Metoda BruteForce(string[] passwords, CancellationToken cancellationToken) wykonuje atak brute force na podane hasła. Iteruje przez tablicę haseł, sprawdzając jednocześnie, czy żądane jest przerwanie ataku poprzez sprawdzenie wartości IsCancellationRequested obiektu CancellationToken. Jeśli żądane jest przerwanie, metoda zwraca informację o przerwaniu. W przeciwnym razie, metoda kontynuuje atak i zwraca znalezione hasła w postaci tekstu. Metoda Cancel(string guid) przyjmuje identyfikator żetonu (tokenu) jako parametr. Na podstawie tego identyfikatora, metoda pobiera odpowiadający mu obiekt CancellationTokenSource z m_TokensCatalog i wywołuje metodę Cancel() dla tego obiektu, co powoduje przerwanie ataku brute force. GetEnumerator() ta metoda implementuje interfejs IEnumerable i zwraca enumerator, który umożliwia iterację po obiektach klasy BruteForceIter. W tej metodzie następuje generowanie permutacji ciągu alphabet dla określonego zakresu wartości od _min do _max. Oblicza ciąg znaków a na podstawie wartości l i power. Wartość l jest dzielona przez długość alphabet, a reszta wskazuje na kolejne znaki w alphabet. Metoda zwraca wynik jako ciąg znaków. SaveResultToFile(string text) metoda zapisuje podany tekst do pliku "passwords2.txt" na pulpicie. Jeśli plik nie istnieje, zostaje utworzony, a tekst jest zapisywany jako nowa linia. Jeśli plik już istnieje, tekst jest dołączany jako nowa linia do istniejącej zawartości pliku. W kontrolerze natomiast mamy udostępnianie trzech akcji, które obsługują żądania HTTP związane z operacjami bruteforce. Wykorzystuje serwis IBruteForceService do wykonywania tych operacji i zwraca odpowiednie wyniki jako odpowiedzi HTTP. Dodatkowo w projekcie "Client" dość istotny jest plik appsettings.json w którym to wpisujemy nasze ip w celu nasłuchiwanie serwera.

2.2 BRUTE FORCE.

```
namespace WindowsFormsApplication1
{
    class BruteForce
    {
        private static String password; //521ab
        public static StringBuilder str = new StringBuilder("");
        private static int min = 32, max = 127;
        private static DateTime start;

        public BruteForce(String password1)
        {
            password = password1;
        }

        public void RunAlghoritm()
        {
            start = DateTime.Now;

            while (true)
            {
                str.Append((char)min);

                for (int i = 0; i < str.Length - 1; i++)
                {
                    for (int j = min; j < max; j++)
                    {
                        str[i] = (char)j;
                        bruteForceAlghoritm(i + 1);
                    }
                }
            }
        }

        public void bruteForceAlghoritm(int index)
        {
            for (int i = min; i < max; i++)
            {
                str[index] = (char)i;

                if (index < str.Length - 1)
                {
                    bruteForceAlghoritm(index + 1);
                }

                //Console.WriteLine(str);

                if (str.ToString().Equals(password))
                {
                    DateTime stop = DateTime.Now;
                    TimeSpan czasWykonania = stop - start;
                    int czasLiczbowy = Convert.ToInt32(czasWykonania.TotalMilliseconds);
                    StringBuilder str1 = new StringBuilder("");
                    str1.Append("" + str);
                    str1.Append("; " + czasLiczbowy + "ms");
                    SaveResultToFile(str1 + "");
                }
            }
        }
    }
}
```

Rys. 2.1 Pierwszy algorytm typu Brute Force.

Cały algorytm mieści się w jednej klasie **BruteForce** i jest uruchamiany za pomocą metody **RunAlghoritm()**. Program jest początkową wersją implementacji, która będzie rozbudowywana w celu poprawy jej obsługi (konfiguracja parametrów) co może mieć znaczący wpływ na zmianę wydajności działania samego algorytmu. Najważniejszymi atrybutami tej klasy są pola: **password**, **min** oraz **max**. Pierwszy z nich służy oczywiście do obsługi samego hasła, natomiast kolejne wyznaczają odpowiednio początek i koniec alfabetu. Są to liczby, ponieważ w tym programie odpowiadają one znakom z tablicy ASCII. Najczęściej używane zostały przedstawione w tabeli poniżej.

DEC	HEX	Znak	DEC	HEX	Znak	DEC	HEX	Znak	DEC	HEX	Znak
0	00	Null	32	20	Spacja	64	40	@	96	60	`
1	01	Start Of Heading	33	21	!	65	41	A	97	61	a
2	02	Start of Text	34	22	"	66	42	B	98	62	b
3	03	End of Text	35	23	#	67	43	C	99	63	c
4	04	End of Transmission	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal Tab	41	29)	73	49	I	105	69	i
10	0A	Line Feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical Tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form Feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage Return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift Out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift In	47	2F	/	79	4F	O	111	6F	o
16	10	Data Link Escape	48	30	0	80	50	P	112	70	p
17	11	Device Control 1 (XON)	49	31	1	81	51	Q	113	71	q
18	12	Device Control 2	50	32	2	82	52	R	114	72	r
19	13	Device Control 3 (XOFF)	51	33	3	83	53	S	115	73	s
20	14	Device Control 4	52	34	4	84	54	T	116	74	t
21	15	Negative Acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous Idle	54	36	6	86	56	V	118	76	v
23	17	End of Transmission Block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of Medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[123	7B	{
28	1C	File Separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group Separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record Separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit Separator	63	3F	?	95	5F	_	127	7F	Delete

Rys. 2.1 Pierwszy algorytm typu Brute Force.

Ponadto możemy dostrzec mierzenie czasu, które rozpoczyna się tuż po wywołaniu samej metody, a kończy w ostatniej instrukcji warunkowej - stwierdzającej czy dane hasło zostało złamane. Czas wykonania zapisywany jest w **milisekundach** w postaci liczbowej. Następnie hasło wraz z wartością czasu operacji jest zapisywane do pliku (możliwe, że format tekstowy będzie podlegał jeszcze modyfikacjom, bądź całkowitym zamianie np. na format typu excel). Sam plik tworzony jest w sposób uniwersalny na pulpicie wykonywanego sprzętu komputerowego. Co pozwala uelastyczyć rozwiązanie i zaoszczędzić czasu na zmiany samych ścieżek, a skupić się na poważniejszych kwestiach obsługi samego programu. Oczywiście jeżeli taki plik już istnieje jest on już tylko nadpisywany. Kod zapisu wyników programu zaprezentowano na poniższym listingu.

```
public void SaveResultToFile(string text)
{
    string path1 = Environment.GetFolderPath(Environment.SpecialFolder.Desktop).ToString() + "\\passwords" + ".txt";

    DateTime thisMoment = DateTime.Now;
    if (!File.Exists(path1))
    {
        using (StreamWriter sw = File.CreateText(path1))
        {
            sw.WriteLine(text);
        }
    }
    else
    {
        using (StreamWriter sw = File.AppendText(path1))
        {
            sw.WriteLine(text);
        }
    }
}
```

Rys. 2.3 Zapis haseł do pliku.

Tak wyglądała pierwsza wersja aplikacji, jednak po pewnych uaktualnieniach, poprzez rozbudowywanie pewnych funkcji np, poprzez możliwość konfiguracji zakresów, czy kontroli ilości wątków nastąpiło wiele zmian. Dzięki nim program pozwolił na lepszą komunikację z użytkownikiem, lepszą czytelność kodu oraz łatwiejszą obsługę samego rozwiązania czy samego testowania, które odbywa się automatycznie BEZ fizycznej konieczności potwierdzania danej operacji - np połączenia z klientem czy uruchomienia samego algorytmu.

Program został maksymalnie zautomatyzowany oraz zoptymalizowany. W celu wykorzystania maksymalnej mocy obliczeniowej komputera, zdecydowano się na wielowątkowe przetwarzanie w programie hamującym MD5.

```
using System;
using System.Collections.Generic;

namespace BruteForceApp
{
    class BruteForce
    {
        public static void Run(List<string> passwords, string alphabet, string startRange, string endRange)
        {
            List<string> listOfPermutations = PermutationsGenerator.GeneratePermutationsWithRepetitions(alphabet, startRange, endRange); //generowanie permutacji

            passwords = MD5.CreateListOfMD5(passwords); //pobiera w parametrze listę haseł, zamienia ją na listę hash'y i zapisuje do tej samej listy
            listOfPermutations = MD5.CreateListOfMD5(listOfPermutations); //jak powyżej ... tylko dla listy permutacji

            ComparePasswordsListWithHashes(passwords, listOfPermutations);
        }

        private static void ComparePasswordsListWithHashes(List<string> passwords, List<string> listOfPermutations)
        {
            foreach (string password in passwords)
            {
                if (CompareHash(password, listOfPermutations))
                {
                    // ----- !!! HASŁO ZNALEZIONE !!! -----
                    Console.WriteLine("Hasło złamane");
                }
            }
        }

        private static bool CompareHash(string password, List<string> listOfPermutations)
        {
            foreach (string permutation in listOfPermutations)
            {
                if (password.Equals(permutation))
                {
                    return true;
                }
            }

            return false;
        }
    }
}
```

Rys. 2.3 Nowy Brute Force.

```

using System;
using System.Collections.Generic;

namespace BruteForceApp
{
    public class PermutationsGenerator
    {
        public static List<string> GeneratePermutationsWithRepetitions(string alphabet, string startRange, string endRange)
        {
            List<string> permutations = new List<string>();

            GeneratePermutationsWithRepetitionsHelper(alphabet, startRange, endRange, "", permutations);

            return permutations;
        }

        private static void GeneratePermutationsWithRepetitionsHelper(string alphabet, string startRange, string endRange, string current, List<string> permutations)
        {
            if (current.Length >= startRange.Length && CompareTwoString(current, startRange) >= 0 && CompareTwoString(current, endRange) <= 0)
            {
                permutations.Add(current);
            }

            if (current.Length >= endRange.Length)
            {
                return;
            }

            for (int i = 0; i < alphabet.Length; i++)
            {
                GeneratePermutationsWithRepetitionsHelper(alphabet, startRange, endRange, current + alphabet[i], permutations);
            }
        }

        private static int CompareTwoString(string s1, string s2)
        {
            char[] charArray1 = s1.ToCharArray();
            char[] charArray2 = s2.ToCharArray();

            int minLength = Math.Min(charArray1.Length, charArray2.Length);

            if (charArray1.Length < charArray2.Length)
                return -1;
            else if (charArray1.Length > charArray2.Length)
                return 1;

            for (int i = 0; i < minLength; i++)
            {
                if (charArray1[i] < charArray2[i])
                    return -1;
                else if (charArray1[i] > charArray2[i])
                    return 1;
            }

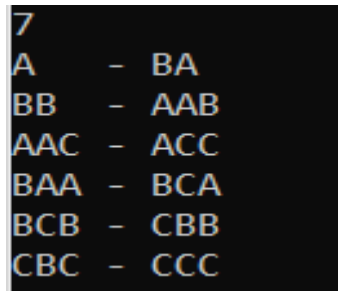
            return 0;
        }
    }
}

```

Rys. Generowanie permutacji na podstawie zakresów.

2.3 OBLICZANIE GRANIC.

Aby obciążenie wszystkich komputerów w rozproszonej wersji projektu było równomierne potrzebowaliśmy odpowiedniej funkcji obliczającej oraz wyznaczającej granice wygenerowanych możliwych kombinacji, aby móc je rozsyłać do pozostałych komputerów(potencjalnych klientów).



```
7
A - BA
BB - AAB
AAC - ACC
BAA - BCA
BCB - CBB
CBC - CCC
```

Rys. Wyniki sugerowanych granic.

Oczywistym jest, że jest to kod programu przedstawiony poniżej pochodzi z pierwotnych wersji aplikacji.

```
public static void GetBoundFromFile(int numberOfComputers)
{
    int numberOfFileLines = File.ReadLines(path).Count();
    int numberOfLines = numberOfFileLines / numberOfComputers;
    Console.WriteLine(numberOfLines + "");

    for (int i = 1; i < numberOfFileLines; i += numberOfLines)
    {
        if(i + numberOfLines - 1 > numberOfFileLines)
        {
            //Console.WriteLine(i + "-" + numberOfFileLines);
            Console.WriteLine(File.ReadLines(path).Skip(i - 1).Take(1).First() + "-" + File.ReadLines(path).Skip(numberOfFileLines - 1).Take(1).First());
        }
        else
        {
            //Console.WriteLine(i + "-" + (i + numberOfLines - 1));
            Console.WriteLine(File.ReadLines(path).Skip(i - 1).Take(1).First() + "-" + File.ReadLines(path).Skip(i + numberOfLines - 2).Take(1).First());
        }
    }
}
```

Rys. Pierwotny kod odczytu granic.

2.4 HASZOWANIE.

W celu zoptymalizowania pracy programu haszującego (w tym programie haszowane są hasła oraz permutacje) zdecydowano się maksymalnie wykorzystać zasoby sprzętowe używając wszystkie dostępne wątki jakie oferuje nam dany client.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace CustomerAPI.Services
{
    class MD5
    {
        public static List<String> CreateListOfMD5(List<String> listOfPasswords)
        {
            List<string> passwordConvertedOnMD5 = new List<string>();

            Parallel.ForEach(listOfPasswords, new ParallelOptions { MaxDegreeOfParallelism = Environment.ProcessorCount }, password =>
            {
                passwordConvertedOnMD5.Add(CreateMD5(password));
            });

            return passwordConvertedOnMD5;
        }

        public static string CreateMD5(string password)
        {
            using (System.Security.Cryptography.MD5 md5 = System.Security.Cryptography.MD5.Create())
            {
                byte[] inputBytes = System.Text.Encoding.ASCII.GetBytes(password);
                byte[] hashBytes = md5.ComputeHash(inputBytes);

                //return Convert.ToHexString(hashBytes); // .NET 5 +

                // Convert the byte array to hexadecimal string
                StringBuilder sb = new System.Text.StringBuilder();
                for (int i = 0; i < hashBytes.Length; i++)
                {
                    sb.Append(hashBytes[i].ToString("X2"));
                }
                return sb.ToString().ToLower();
            }
        }
    }
}
```

Rys. Kod współbieżnego haszowania .

2.4 TESTOWANIE.

Przy testowania aplikacji nie sposób pominąć doboru parametrów ziarnistości, przyspieszenia oraz efektywności. Poniżej podano najważniejsze wzory dzięki, którym przeprowadzono testy porównawcze, aby lepiej poznać jego działanie, właściwości oraz samą wydajność na różnej ilości hostów porównując różne rzędy ilości haseł.

Ziarnistość

$$G = \frac{czas_{obliczen}}{czas_{komunikacji}}$$

Klasy równoległości

- Równoległość drobnoziarnista (małe G)
- Równoległość gruboziarnista (duże G)

Rys. Wzór na ziarnistość^[22].

Przyspieszenie

- $T_o(1)$ – optymalny czas jednoprocessorowego rozwiązania
- $T(p)$ – czas wykonania zadania przez p procesorów

$$S(p) = \frac{T_o(1)}{T(p)}$$

Legenda:

p - ilość procesorów

$T(p)$ - czas wykonywania zadania dla p procesorów

$S(p)$ - przyspieszenie dla p procesorów

Rys. Wzór na przyspieszenie^[22].

Efektywność

$$E(p) = \frac{S(p)}{p} = \frac{T_o(1)}{T(p) * p}$$

W idealnym przypadku $S(p)=p$ i $E(p)=1$

Legenda:

p - ilość procesorów

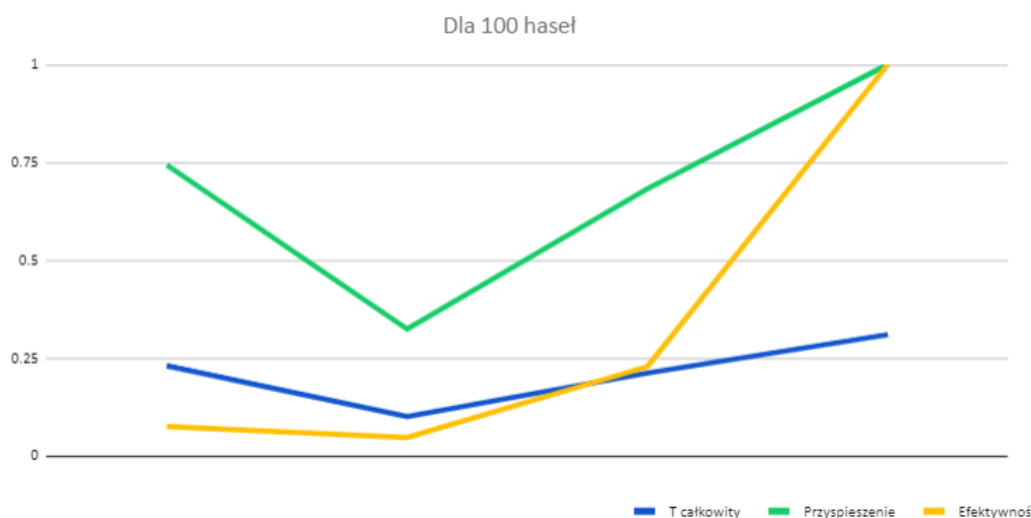
T(p) - czas wykonywania zadania dla p procesorów

E(p) - efektywność dla p procesorów

S(p) - przyspieszenie dla p procesorów

Rys. Wzór na efektywność [22].

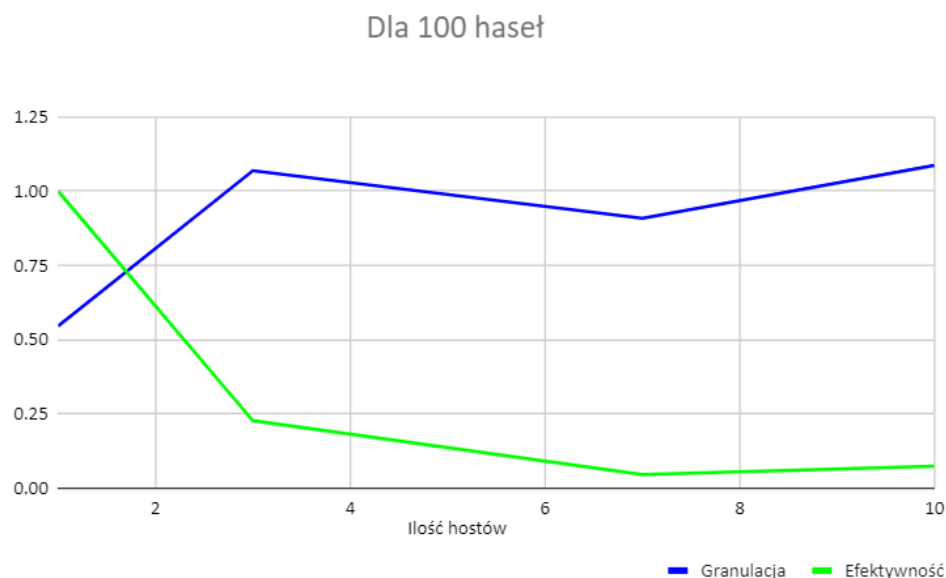
Na podstawie podanych powyżej wzorów postanowiono przeprowadzić kolejne testy wydajnościowe. Wykonano szereg prób kontrolując zarówno czas obliczeń, czas komunikacji komputerów, jak i czas wykonywania programu dla poszczególnych komputerów. Ze względu na obszerność wyników postanowiono przedstawić jedynie część poglądową dla dwóch ilości przesyłanych danych: dla 100 oraz dla 10.000 haseł. Poniższe wykresy przedstawiają wyniki dla kolejnych ilości.



Rys. Wykres porównania przyspieszenia i efektywności dla 100 haseł.

Ilość hostów	10	7	3	1
T całkowity	0.22954	0.100003	0.2108954	0.3091307
Przyspieszenie	0.7425338215	0.3234974721	0.6822208212	1
Efektywność	0.07425338215	0.04621392459	0.2274069404	1
Granulacja	10	7	3	1

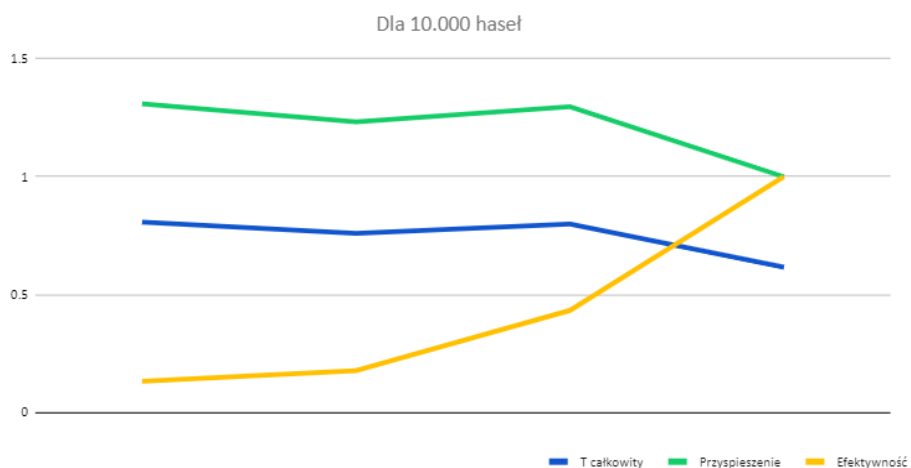
Rys. Tabela porównawcza przyspieszenia i efektywności dla 100 haseł.



Rys. Wykres porównania granularności i efektywności dla 100 haseł.

Ilość hostów	10	7	3	1
Granulacja	1.086670364	0.909085124	1.068668745	0.545895
Efektywność	0.074253382	0.046213925	0.22740694	1

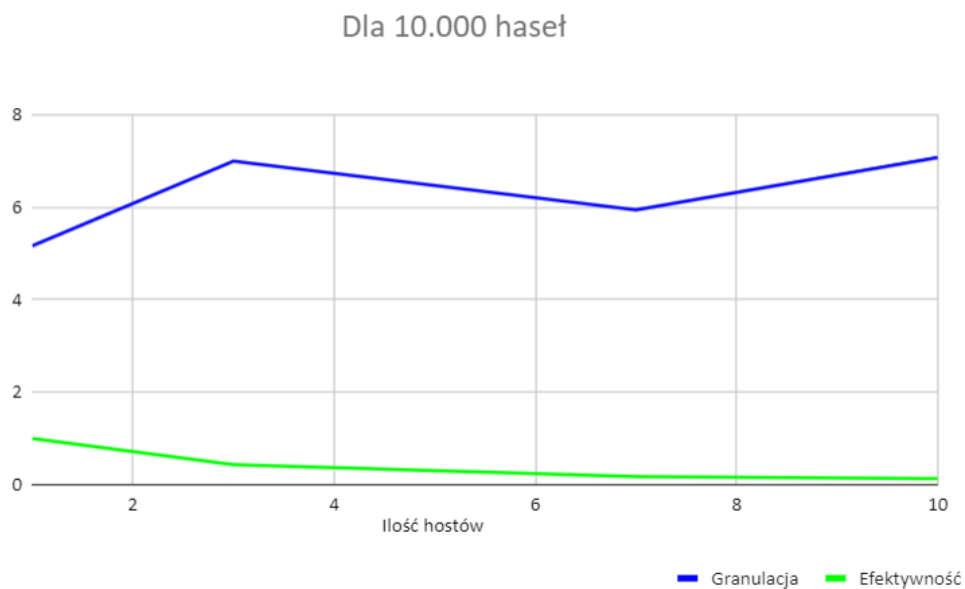
Rys. Tabela porównania granularności i efektywności dla 100 haseł.



Rys. Wykres porównania przyspieszenia i efektywności dla 10.000 haseł.

Ilość hostów	10	7	3	1
T całkowity	0.8070241	0.759581	0.7989263	0.6159994
Przyspieszenie	1.310105335	1.23308724	1.296959542	1
Efektywność	0.1310105335	0.17615532	0.4323198475	1

Rys. Tabela porównawcza przyspieszenia i efektywności dla 10.000 haseł.



Rys. Wykres porównania granulacji i efektywności dla 10.000 haseł.

Ilość hostów	10	7	3	1
Granulacja	7.069918203	5.938206763	6.989263	5.159994
Efektywność	0.131010533	0.17615532	0.432319847	1

Rys. Tabela porównania granulacji i efektywności dla 10.000 haseł.

LITERATURA

[1] Algorytm brute force

http://mumin.pl/Skrypt_A_do_Z/lekcje/l06.html

[2] Działanie algorytmu brute force

<http://prac.im.pwr.wroc.pl/~szwabin/assets/algo/lectures/5.pdf>

[3] Opis ataków słownikowych.

<https://sekurak.pl/crackstation-udostepnia-ogromny-sloownik-hasel/>

[4] Kodowanie słownikowe

<https://www.ire.pw.edu.pl/~rois/dydaktyka/skrypt/5.pdf>

[5] Działanie metody siłowej

http://neo.dmcs.p.lodz.pl/ptda/wyklady/wyklad_5.pdf

[6] Równoległe łamanie haseł

<http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-article-BGPK-3546-3483>

[7] Schemat blokowy algorytmu słownikowego

<https://slideplayer.pl/slide/11339059/>

[8] Równoległe łamanie haseł - przykład

https://repozytorium.biblos.pk.edu.pl/redo/resources/32576/file/suwFiles/PlazekJ_RownolegleLamanie.pdf

[9] Brute force/metoda słownikowa

<https://docplayer.pl/8991975-Akd-metody-sloownikowe.html>

[10] Kod algorytmu brute force

<https://www.dotnetoffice.com/2022/10/brute-force-algorithm-in-c.html>

[11] Kod algorytmu brute force - 2.

<https://gist.github.com/jwoschitz/1129249>

[12] Kod algorytmu brute force - 3.

<https://www.c-sharpcorner.com/article/how-using-brutal-force-could-improve-code-quality>

[13] Wykres doboru ziarnistości.

https://achilles.tu.kielce.pl/portal/Members/332bd2d158a24964b2b98a7fc2879842/programowanie-wspolbiezne/wyklady/1-wstep/wyk1_wprowadzenie.pdf

[14] Tabela transformacji do uzyskania określonej ziarnistości.

http://zbc.ksiaznica.szczecin.pl/Content/4621/PHD_Krzysztof_Siedlecki.pdf

[15] Haszowanie

<https://bezpieczny.blog/co-to-jest-hash/>

[16] SHA

<https://sekurak.pl/kompendium-bezpieczenstwa-hasel-atak-i-obrona/#:~:text=SHA2%20istnieje%20w%20czterech%20odmianach,-384%20oraz%20SHA-512.>

[17] SHA-256

<https://sha256algorithm.com>

[18] Hashowanie

http://wojciech.bozejko.staff.iiar.pwr.wroc.pl/elearning/Wyk5_tabl_hash.pdf

[19] Twierdzenie wariacji z powtórzeniami

<https://opracowania.pl/opracowania/matematyka/wariacje-z-powtorzeniami,oid,2046>

[20] Opis rockyou.txt

<https://en.wikipedia.org/wiki/RockYou>

[21] Opis rockyou.txt

<https://stackoverflow.com/questions/9290498/how-can-i-limit-parallel-foreach>

[22] Wzory granulacji, przyspieszenia, efektywności

https://achilles.tu.kielce.pl/portal/Members/332bd2d158a24964b2b98a7fc2879842/programowanie-systemow-rozproszonych/wyklady/wyklad-1/wyk1_wprowadzenie.pdf/view